(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) EP 0 961 204 A2

(12) EUROPEAN PATENT APPLICATION

(43) Date of publication:
01.12.1999 Bulletin 1999/48

(51) Int. Cl.[6]: G06F 9/46

(21) Application number: 98123604.5

(22) Date of filing: 10.12.1998

(84) Designated Contracting States:
AT BE CH CY DE DK ES FI FR GB GR IE IT LI LU MC NL PT SE
Designated Extension States:
AL LT LV MK RO SI

(30) Priority: 28.05.1998 US 86147

(71) Applicant:
Hewlett-Packard Company
Palo Alto, California 94304 (US)

(72) Inventor: Williams, Roy H.
Loveland, Colorado 80538 (US)

(74) Representative:
Schoppe, Fritz, Dipl.-Ing.
Schoppe, Zimmermann & Stöckeler
Patentanwälte
Postfach 71 08 67
81458 München (DE)

(54) Thread based governor for time scheduled process execution

(57) A scheduling governor that regulates the number of scheduled tasks that are executed concurrently over a network computer system is presented. All task requests that are scheduled to be executed within a pre-specified interval of time (710), are serviced according to their priority. During heavy load times, the scheduling governor prevents overloads of the processing resources of the host computer by limiting the number of concurrently executing scheduled tasks to a pre-specified capacity dimension (742). Task requests that are unable to be run due to the governed cap (742) on the number of allowed concurrently executing processes are given a priority to be executed once one of the fixed number of execution slots becomes available. Accordingly, the scheduling governor allows each scheduled task to be executed as close to its scheduled time as possible yet prevents system resource overload to improve efficiency and performance.

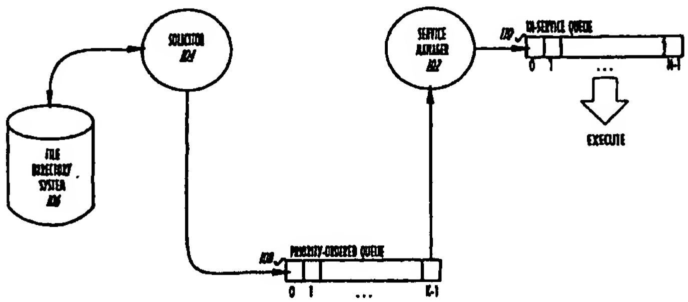

FIG. 1

EP 0 961 204 A2

Printed by Xerox (UK) Business Services
2.16.7/3.6

Description

Field of the Invention

[0001] The present invention pertains generally to network management systems, and more particularly, to a method and system for governing the number of concurrently executing processes in a process scheduler.

Background of the Invention

[0002] The maintenance of a computer network is typically performed under the control of a network management system. The network management system monitors the occurrences of certain events across the system, such as network node utilization and disk I/O performance. To assist in network maintenance, reporting applications may be used to organize and present network event information into a format useful to the system administrator. These reports, depending on the type of information gathered, may be generated on a pre-determined schedule, e.g., once every ten minutes, once daily, or once weekly.

[0003] In the prior art, scheduling applications that schedule the execution of processes such as report generating applications causes execution of the scheduled applications to occur at precisely the time that they are scheduled. This technique can be problematic. Due to the cyclical nature of work week timing, there is a very high propensity for requests to be clustered around certain times of the day. For example, it often makes sense to generate daily reports at exactly midnight so that data used in generating those reports need not necessarily be saved from day to day. When too many processes are scheduled at the same time, and hence concurrently run, the performance of the network system may be adversely impacted due to not enough system resources being available to service all of the executing processes. System resources must then be time-shared according to a priority scheme across all executing processes, resulting in very slow performance for all processes or at least those that have low priority.

[0004] Accordingly, a need exists for a method for governing the number of allowed concurrently executing processes launched by the scheduler, to thereby increase system performance in the face of a heavy schedule load.

Summary of the Invention

[0005] The present invention is a thread-based scheduling governor that regulates the number of scheduled tasks that are executed concurrently. The schedule governor of the invention is implemented using threads. In the system of the invention, a task goes through a life cycle. In infant form, it is given a slot on the file system as a "request file"; it graduates from "in-file" form to "in-memory" form, where it is maintained as an idle thread in a priority-ordered queue. When the time comes for the task to be executed, the task is allotted a space, if available, in a capped in-service queue, where the task graduates from "in-memory" form to "process execution" form. When a task is entered in the in-service queue, the thread launches the process(es) to accomplish the task, and monitors them for their completion. Upon completion of the monitored processes, the thread cleans up its in-service queue entry to allow another pending task to occupy the regulated cap on dispatched processes. Accordingly, a thread maintains control over a task, but system resources do not become dedicated to the processes required by the task until a process execution slot in a capacity-governed in-service queue becomes available.

[0006] In accordance with the invention, all task requests that are scheduled to be executed within a pre-specified interval of time are serviced on a first-in first-out (FIFO) basis and use absolute, or "epoch", time according to their scheduled position in time. During heavy load times, the scheduling governor prevents overloads of the processing resources of the host computer by limiting the number of concurrently executing processes to a pre-specified capacity dimension. Task requests that are unable to be run due to the governed cap on the number of allowed concurrently executing processes are deferred until one of the fixed number of execution slots becomes available. Accordingly, the scheduling governor allows each scheduled task to be executed as close to its scheduled time as possible yet prevents system resource overload to improve efficiency and performance.

Brief Description of the Drawing

[0007] The invention will be better understood from a reading of the following detailed description taken in conjunction with the drawing in which like reference designators are used to designate like elements, and in which:

FIG. 1 is a block diagram of a processing system in which the scheduling governor of the invention operates;

FIG. 2 is a flowchart describing the steps performed by the governed scheduling system of the invention;

FIG. 3 is a diagram of the structure of an example file directory existing on a system;

FIG. 4 illustrates an example format of a request file;

FIG. 5 is a data model of one preferred embodiment of the invention;

FIG. 6 is a flowchart illustrating the steps performed by a search utility that registers schedule requests with the scheduler process; and

FIG. 7 is a flow diagram of the method of operation of the data model of FIG. 5.

Detailed Description

[0008] FIG. 1 is a block diagram of a processing system 100 in which the scheduling governor of the invention operates. System 100 includes a service manager 102, a solicitor function 104, a file directory system 106, a Priority-Ordered queue 108, and an In-Service queue 110. Solicitor function 104 locates and queues task requests into Priority-Ordered queue 108 as specified in file directory system 106 which will be due to be executed within a time interval specified by service manager 102. Service manager 102 only accepts task requests from Priority-Ordered queue 108 that are due within the specified time interval. All other task requests are either discarded until discovered on a later search by solicitor function 104. Task requests that are discovered by solicitor function 104 to be scheduled within the pre-specified time interval are entered into Priority-Ordered queue 108 in sorted time order, where one end of the queue represents the current time and the other end of the queue represents the length of the time interval before the current time. Thus, tasks are transferred from Priority-Ordered queue 108 to In-Service queue 110 on a first-in first-out (FIFO) basis. Service manager 102 manages an In-Service queue 110. In-Service queue includes *n* execution entries, where *n* is the pre-specified maximum number of scheduled tasks that are allowed to be executed concurrently. When any execution entry in In-Service queue is empty (i.e., becomes available), service manager 102 transfers the highest priority (in the preferred embodiment, priority is time-ordered) task request from priority-ordered queue to the available execution entry in In-Service queue, and execution of the task request commences. When a task request completes execution, it is removed from the In-Service queue, and if its next execution time does not exceed the pre-specified time interval of the solicitor function 104, is re-entered into Priority-Ordered queue 108 in sorted position. If its next execution time does exceed the pre-specified time interval, the task request is discharged from memory and its lifecycle completes.

[0009] FIG. 2 is a flowchart describing the steps performed by the governed scheduling system of the invention. As shown, the steps performed include a first step 202 of discovering a task request that is due to be executed within the pre-specified time interval. Task requests that are discovered to be scheduled within the pre-specified time interval are entered into a priority-ordered queue 108 in sorted time order in step 204. In-Service queue 110 is monitored in step 206 until an execution entry in In-Service queue becomes available. Once an available execution entry is discovered, the highest priority task request from priority-ordered queue is transferred to the available execution entry in In-Service queue in step 208. This is contingent on its next time due being less than or equal to the current time. In the preferred embodiment, the priority of a task request is measured in terms of time. In step 210, execution of the task request commences. Steps 202 through 210 are repeated continuously.

[0010] When a task request completes execution, as determined in step 212, it is removed from the In-Service queue in step 214. It is determined in step 216 whether its next execution time exceeds the pre-specified time interval. If its next execution time does not exceed the pre-specified time interval, the task request is re-entered into priority-ordered queue in sorted position in step 204, and the cycle is repeated. If its next execution time does exceed the pre-specified time interval, the task request is discharged from memory in step 220.

[0011] FIG. 3 is a diagram of the structure of an example file directory tree 300 existing in file directory system 106. File directory tree 300 illustrates the hierarchical storage of individual request files in the network file system. Each leaf directory, labeled in the illustrative embodiment as "Router", "Nodes", "Site", "Subnet", "Events by Severity", and "Threshold Violations", contains a request file, each of a name recognizable by the search utility (i.e., Solicitor function 104), labeled "Request_File" in FIG. 3, that contain command lines for executing a particular task, and an accompanying schedule, discussed in FIG. 4, indicating when to execute the task on a periodic basis. Each request file is accompanied, in the same directory where the request file is found, by all files necessary to execute the request.

[0012] FIG. 4 illustrates an example format of a request file 400 that resides in file directory system 106. In accordance with one embodiment of the request file format, the task to be performed and schedule for performing the task are indicated according to a predefined format. A schedule is placed on a line, e.g., 402; a command line executable program is placed on one or more next lines 404, and if required for execution, a command constructor code is placed on a line 406 following the command lines 402, and a command destructor code is placed on a line 408 following line 406 if it exists or lines 402 otherwise. In the preferred embodiment, the request file must include a schedule line 402 and a command line 404; however, their position within the file is not fixed. A set of utilities parses request file 400 into a set of name/value/attribute tuples of information. In the illustrative embodiment, the name/value/attribute tuple of line 402 of request file 400 would yield:

Name =	"schedule"
Value =	"10 0-23 ****"
Attrubute =	"10 min after every hour"

[0013] A set of access control utilities is also provided to create a semaphore file adjacent to the request file being parsed. Cooperating programs and treads yield to the presence of the semaphore file. Solicitor 520 and Service Manager 516, discussed with respect to FIG. 5,

are examples of threads which cooperate.

[0014] FIG. 5 is a data model 500 of the implementation of a preferred embodiment of the invention. Data model 500 is a set of defined classes, implemented in the Java® programming language in the preferred embodiment, that are dynamically instantiated for the lifetime of the scheduling thread.

[0015] Scheduler 502 is the starting point for DispatcherQuery 508 navigation. As DispatcherClient 506 objects make method calls on method server_socket of Scheduler 502, they are serviced by DispatchedThread 510 to perform the actual query in a thread within Scheduler 502. DispatcherClients 506 are able to obtain information as to the socket address port of their Scheduler 502, send it a class name and arguments to load a DispatcherQuery 508 on the server side, and with for the results to come back through the standard output "stdout". DispatcherThreads 510 provide a tread of execution for DispatcherQueries 508 to perform their work within the process space fo Scheduler 502. DispatcherThread 510 is itself an instance of DispatcherQuery 508, but is unique in the sense that it represents the head of the query. DispatcherQuery 508 is a query to be performed on Scheduler 502 identified to it through the DispatcherThread 510.

[0016] ServiceManager 516 is an abstract base class that provides the thread but relys on its derived implementations to know what file they seek and "know how" on how to service them. Once request files are discovered that contain task requests for this object to perform, they are placed in a PriorityOrderedQueue 550 to await dispatch to InService Queue 548 for execution.

[0017] Solicitor 520 is an abstract base class that always comes paired with a derived class of ServiceManager 516. As ServiceManagers 516 are instantiated, they are given an opportunity to load their Solicitor 520. A Solicitor 520 searches the filesystem for a request file of a predefined name agreed upon with its ServiceManager 516. Solicitor 520 typically looks for files that need to begin their program execution within a specified look_ahead_interval.

[0018] FIG. 6 is a flowchart illustrating the steps performed by a solicitor 520 that registers schedule requests with the scheduler process. As shown, solicitor 520 traverses the file directory tree in a step 602, typically using a depth-first search, looking for request files. When a request file is found, in step 604 solicitor 520 registers the task request specified in the contents of the request file with service manager 516. In the preferred embodiment, solicitor 520 registers the task request by determining if the task request is due to be scheduled within a specified interval of time, and, if so, placing the task request in a priority ordered queue 550. The solicitor 520 determines whether any portions of file directory tree remain to be searched in a step 606, and repeats steps 602 and 604 until the entire file directory system 106 has been searched. In one implementation, a solicitor function is employed to seek out request files, all of a specific given name (e.g., request.properties) from a base directory specified at runtime. For example, with reference to file directory tree 300, the solicitor function may be instructed to begin its search from directory "Base Directory". Solicitor function searches the tree, using a depth-first search algorithm, for files having the name "request.properties". When a file of this name is encountered, solicitor 520 parses the contents of the file to retrieve the schedule and command line for the process to be scheduled.

[0019] RequestManager 556 is a derived implementation of abstract base class ServiceManager 516. RequestManager 556 manages a set of ServiceQueue 546 known only to it and not to its base class, which assists it in keeping track of what state each job it is servicing is in. Some jobs may be executing (i.e., exist in the InServiceQueue 548) while others may be waiting in line to be performed (i.e., exist in the PriorityOrderedQueue 550).

[0020] ServiceQueue 546 is an abstract interface. ServiceQueue 546 requires that queues implemented to service RequestManager 556 know how to properly add and remove entries in the RequestManager 556. As entries are manipulated, ServiceManager 516 must be notified of the changes as it keeps a list of current_requests known in all queues. Both InServiceQueue 548 and PriorityOrderedQueue 550 are derived from this object to enforce correct management of ScheduleEntries 540.

[0021] TimeEntry 536 is an object/data structure which defines a time schedule in segmented characteristics which complement the defined format of the request file.

[0022] ScheduleEntries 540 are maintained in ServiceQueues 546. ScheduleEntries 540 represent place holders for Requests 532. A ScheduleEntry 540 can be queried at any time to deterermine when it is "due" to be scheduled. The one incarnation of a ScheduleEntry 540 may be used to load and execute a unique Request 532 on the file system. In other words, the name of the ScheduleEntry 540 is the location of one unique Request 532 on the file system that may be manipulated as many times as necessary. Requests 532 are lockable objects which have persistence in files of the file systems. As they are loaded through their constructors, a lock is obtained through its base class FileAccessCtlr 528. This safety measure is necessary in a multi-threaded and multi-process environment seeking to modify and have sole access rights. Requests 532 may be loaded to examine their contents or execute the command based on its prescribed schedule.

[0023] FIG. 7 is a flow diagram of the method of operation 700 of data model 500. Solicitor 520 searches for request files. When a request file 702 is found, the request file 702 is parsed to extract the schedule and command line to be executed. The "next_time_due", which is the next time the command is scheduled to be executed, is calculated in step 704. If the request pro-

duced in step 704 meets the time criteria required for inclusion in the TimeOrderedQueue 714, as determined in step 705, its ScheduleEntry (i.e., abbreviated form of the Request) is added to TimeOrderedQueue714 in step 708. In step 705, the next_time_due is compared to a lookahead interval to determine whether the ScheduleEntry 706 should be added to the PriorityOrderedQueue 714. The lookahead interval 710 is a pre-specified time interval. Solicitor 520 is a thread that wakes up every lookahead interval, discovers request files on the portion of the system file directory that it is responsible for, and then goes to sleep again until the beginning of the next lookahead interval. Solicitor 520 repeats this cycle as long as the scheduling governor of the invention is running. If, in step 708, the next_time_due for the ScheduleEntry 706 is within the pre-specified lookahead interval, a priority in terms of time is associated with the ScheduleEntry 706 and it is added to the PriorityOrderedQueue 714. In the preferred embodiment, PriorityOrderedQueue 714 is a first-in first-out (FIFO) queue sorted in order of time, and the priority associated with a given ScheduleEntry 706 is indicated simply by the position of the ScheduleEntry 706 in the PriorityOrderedQueue 714. In other words, the ScheduleEntries 540 contained in PriorityOrderedQueue 714 are positioned in the order in which they are to be launched by ServiceManager 516. If the next_time_due for the ScheduleEntry 706 is not due within the pre-specified lookahead interval, it is discarded from memory, or alternatively, is restored to its sorted position in the PriorityOrderedQueue 714.

[0024] ServiceManager 516 is a thread that wakes up and executes on a schedule, e.g., every minute on the minute. When ServiceManager 516 wakes up, it compares the current time, or launch time 720, with the next_time_due of the ScheduleEntries 540 contained in PriorityOrderedQueue 714. If the launch time 720 matches the next_time_due of a ScheduleEntry 540 contained in the time-ordered queue 714, ServiceManager 516 creates an instance 722 of a Request 532 in a step 718. Service_manager method communicates in steps 724 with ServiceQueue 546 to add Request 722 to InServiceQueue 750. If InServiceQueue 750 has an available execution entry, Request 722 is added to InServiceQueue 750 in step 748. If InServiceQueue 750 is full, meaning that the maximum number of concurrent processes are currently executing, Request 722 is returned to TimeOrderedQueue 714. In the alternative, Request 722 is added to a DeferredQueue 728. Just before ServiceManager 516 thread goes to sleep, it restores any existing Requests 722 in DeferredQueue 728 to PriorityOrderedQueue 714. In this embodiment, Requests 722 that are unable to be added to InServiceQueue prior to their being due, do not get executed. Accordingly, arbitration of which Requests get executed when too many Requests are clustered around the same time due is left to the system administrator or some other method of negotiation between competing interests.

[0025] When Requests 722 are added to InServiceQueue 750, a Request thread is launched to execute the command associated with the Request 722. The Request thread monitors the process that it has launched, and once the process completes, the Request thread calls a request complete function in step 734, which captures the process' exit code, standard output file and standard error file. During step 734, the Request 722 is given an opportunity to be fed back into PriorityOrderedQueue 714 by determining in step 708 whether its next_time_due is within the current lookahead interval. If it is not, the lifecycle of the thread of Request 722 ends. In step 738 a remove function of ServiceQueue 546 removes the Request 722 from InServiceQueue 750, which makes available an execution entry in InServiceQueue 750 for another Request 722 from PriorityOrderedQueue 714.

[0026] When the scheduling governor is first started up, InServiceQueue 750 is created in step 744, having a size specified by a user-defined capacity dimension 742. Capaicty dimension 742 operates as the governed cap on number of concurrently executing processes allowed at any given time. Accordingly, with respect to FIG. 1, the capacity dimension 742 is set to N, and In Service Queue 110 comprises N execution entries.

[0027] It will be appreciated from the above detailed description that the scheduling governor of the invention provides an effective method for limiting the number of simultaneously scheduled task requests to a maximum number of concurrently executing task requests.

[0028] Although the invention has been described in terms of the illustrative embodiments, it will be appreciated by those skilled in the art that various changes and modifications may be made to the illustrative embodiments without departing from the spirit or scope of the invention. It is intended that the scope of the invention not be limited in any way to the illustrative embodiment shown and described but that the invention be limited only by the claims appended hereto.

Claims

1. A task scheduling governor for a computer processing system comprising a plurality of tasks to be scheduled to execute according to a respective associated schedule, comprising:

    a priority-ordered execution queue (108, 714) comprising at least one task request, said at least one task request comprising only said tasks that are due to be executed within a pre-specified time interval (710), each said task requests in said priority-ordered execution queue (108, 714) comprising an associated priority indicating a priority of said task request;
    an In-Service queue (110, 750) comprising a plurality of execution entries (0, 1, ..., N-1),

each said execution entry for holding a task request that is commenced when said task request is added to said In-Service queu (110, 750), said task request being removed from said In-Service queue (110, 750) upon completion of said task request to make said associated execution entry available, said In-Service queue (110, 750) limited to a pre-specified number (742) of said execution entries;

a service manager (102) which monitors said In-Service queue (110, 750) and transfers said task request present in said priority-ordered execution queue (108, 714) having said associated priority of highest rank from said priority-ordered execution queue (108, 714) to said In-Service queue (110, 750) when said In-Service queue (110, 750) comprises an available execution entry and said priority-ordered execution queue (108, 714) is not empty; and

a solicitor function (104) which discovers scheduled task requests that are due to be executed within said pre-specified time interval (710) and enters said discovered scheduled task requests into said priority-ordered execution queue (108, 714).

2. A task scheduling governor in accordance with claim 1, wherein: said priority of said task request is time-ordered priority.

3. A task scheduling governor in accordance with claim 2, wherein:

said priority-ordered execution queue (108, 714) comprises a time-ordered execution queue wherein said task requests residing in said time-ordered queue are arranged in order of priority in time.

4. A task scheduling governor in accordance with claim 1, 2 or 3, comprising:

a deferred task queue (728), said deferred task queue (728) comprising said task requests due to be executed but not being is said In-Service queue (110, 750).

5. A method for limiting a number of concurrently executing scheduled tasks in a computer system to a pre-specified capacity dimension (742), said method comprising:

discovering (202) a task request that is due to be executed within a pre-specified time interval (710);

entering (204) said discovered task request into a priority-ordered execution queue (108, 714), said discovered task request having a priority associated with it;

monitoring (206) an In-Service queue (110, 750), said In-Service queue (206) comprising a plurality of execution entries equal in number to said capacity dimension (742), each of said plurality of execution entries comprising either an available execution entry or an In-Service execution entry, each said In-Service execution entry comprising a task currently being executed by said computer system;

when said In-Service queue (110, 750) comprises an available execution entry, transferring (208) said task request having a highest priority from said priority-ordered queue (108, 714) to said available execution entry in said In-Service queue (110, 750);

commencing execution (210) of said transferred task request;

repeating said discovering task through said repeating step.

6. A method in accordance with claim 5, comprising:

when said task request in one of said execution entries completes execution, removing (214) said completed task request from said execution entry in said In-Service queue (110, 750), said execution entry becoming an available execution entry.

7. A method in accordance with claim 5 or 6, comprising:

determining (216) whether a next time due of said removed task request exceeds said pre-specified time interval (710);

if said next time due does not exceed said pre-specified time interval (710), re-entering said task request into said priority-ordered queue (108, 714).

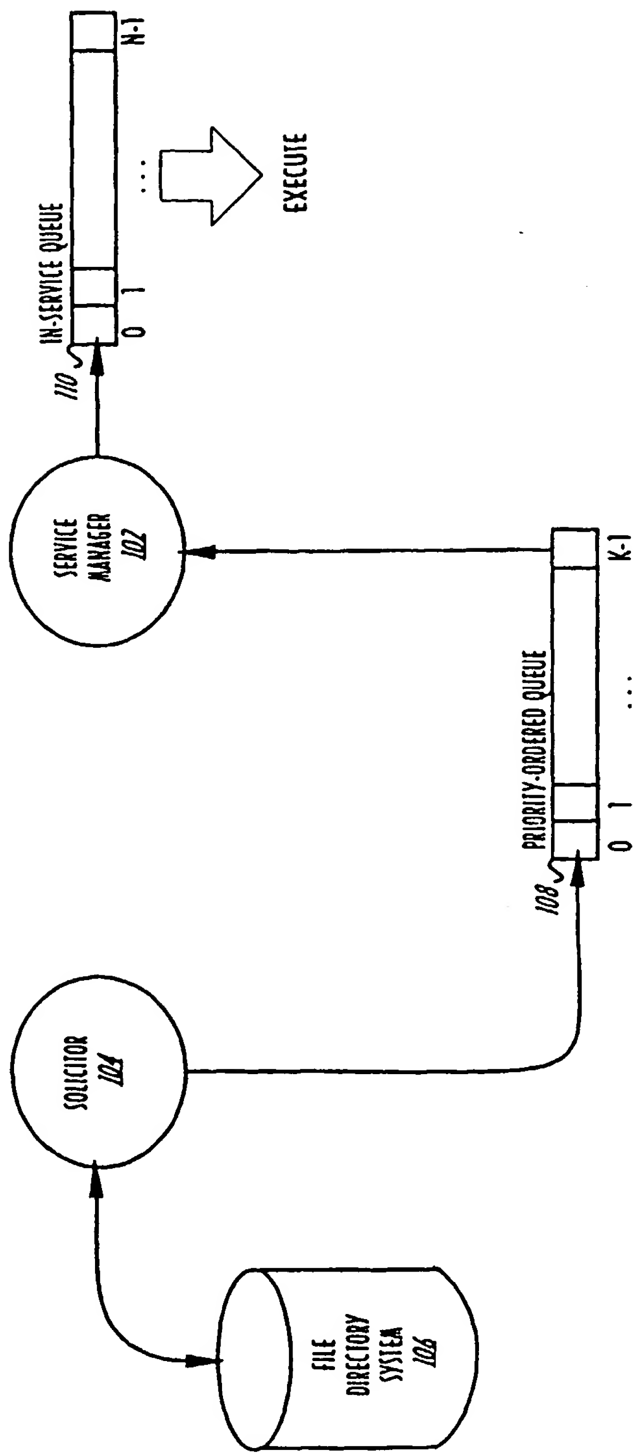
IN-SERVICE QUEUE
110
0
1
N-1
EXECUTE
SERVICE MANAGER 102
PRIORITY-ORDERED QUEUE
108
0
1
K-1
SOLICITOR 104
FILE DIRECTORY SYSTEM 106

FIG. 1

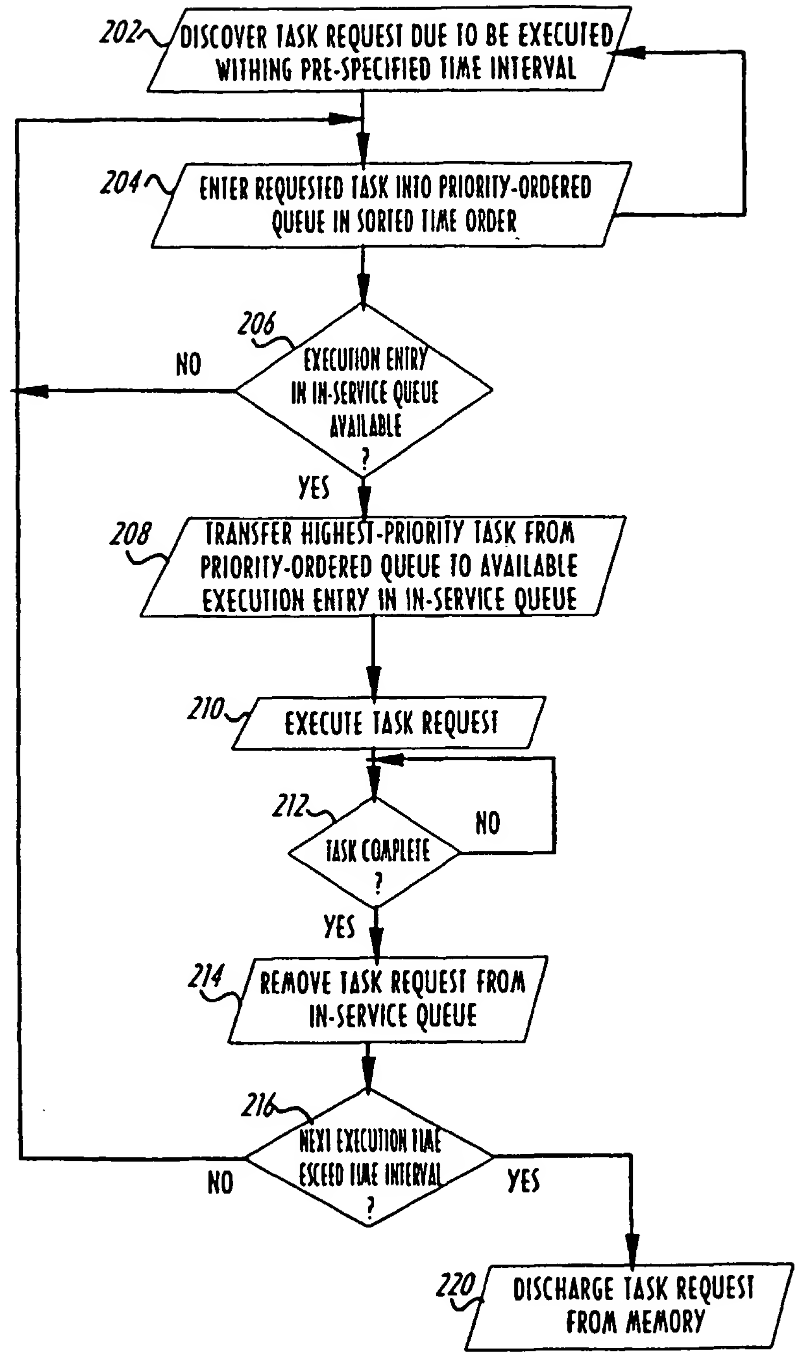
202
DISCOVER TASK REQUEST DUE TO BE EXECUTED WITHING PRE-SPECIFIED TIME INTERVAL
204
ENTER REQUESTED TASK INTO PRIORITY-ORDERED QUEUE IN SORTED TIME ORDER
206
EXECUTION ENTRY IN IN-SERVICE QUEUE AVAILABLE ?
NO
YES
208
TRANSFER HIGHEST-PRIORITY TASK FROM PRIORITY-ORDERED QUEUE TO AVAILABLE EXECUTION ENTRY IN IN-SERVICE QUEUE
210
EXECUTE TASK REQUEST
212
TASK COMPLETE ?
NO
YES
214
REMOVE TASK REQUEST FROM IN-SERVICE QUEUE
216
NEXT EXECUTION TIME ESCEED TIME INTERVAL ?
NO
YES
220
DISCHARGE TASK REQUEST FROM MEMORY

FIG. 2

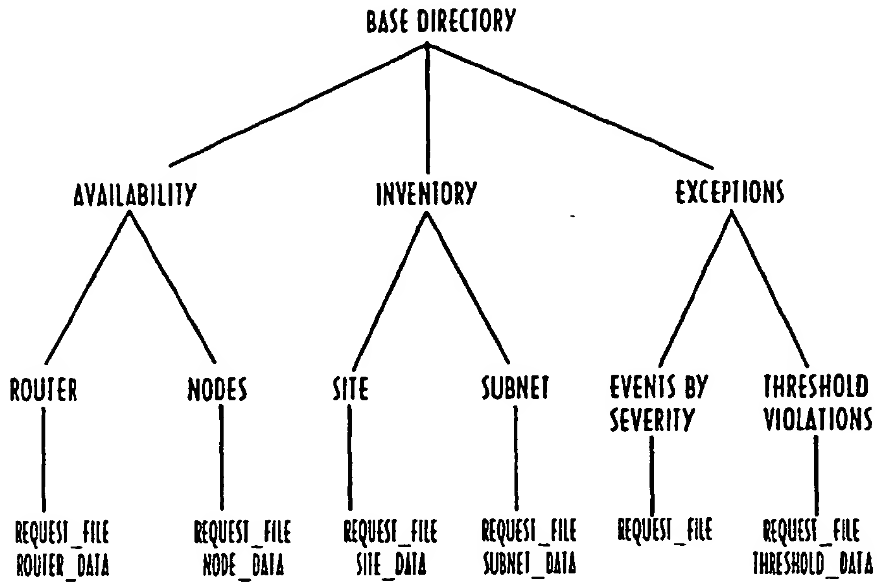
BASE DIRECTORY
AVAILABILITY
INVENTORY
EXCEPTIONS
ROUTER
NODES
SITE
SUBNET
EVENTS BY SEVERITY
THRESHOLD VIOLATIONS
REQUEST_FILE
ROUTER_DATA
REQUEST_FILE
NODE_DATA
REQUEST_FILE
SITE_DATA
REQUEST_FILE
SUBNET_DATA
REQUEST_FILE
REQUEST_FILE
THRESHOLD_DATA

FIG. 3

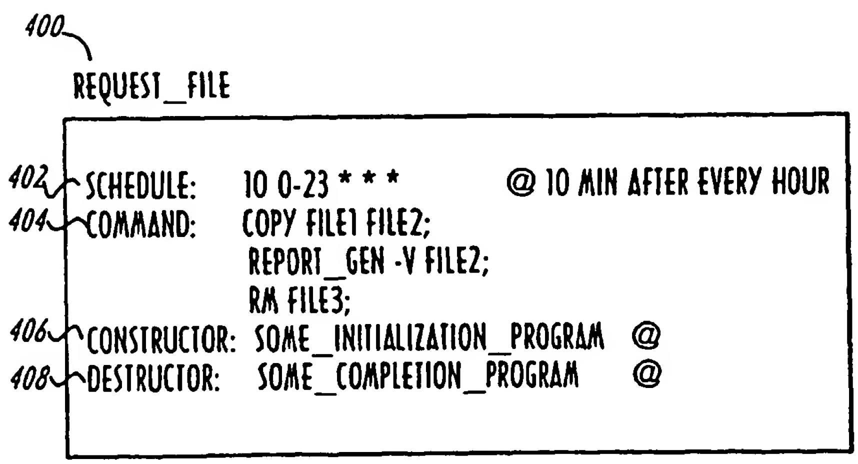
400
REQUEST_FILE
402 SCHEDULE: 10 0-23 * * * @ 10 MIN AFTER EVERY HOUR
404 COMMAND: COPY FILE1 FILE2;
REPORT_GEN -V FILE2;
RM FILE3;
406 CONSTRUCTOR: SOME_INITIALIZATION_PROGRAM @
408 DESTRUCTOR: SOME_COMPLETION_PROGRAM @

FIG. 4

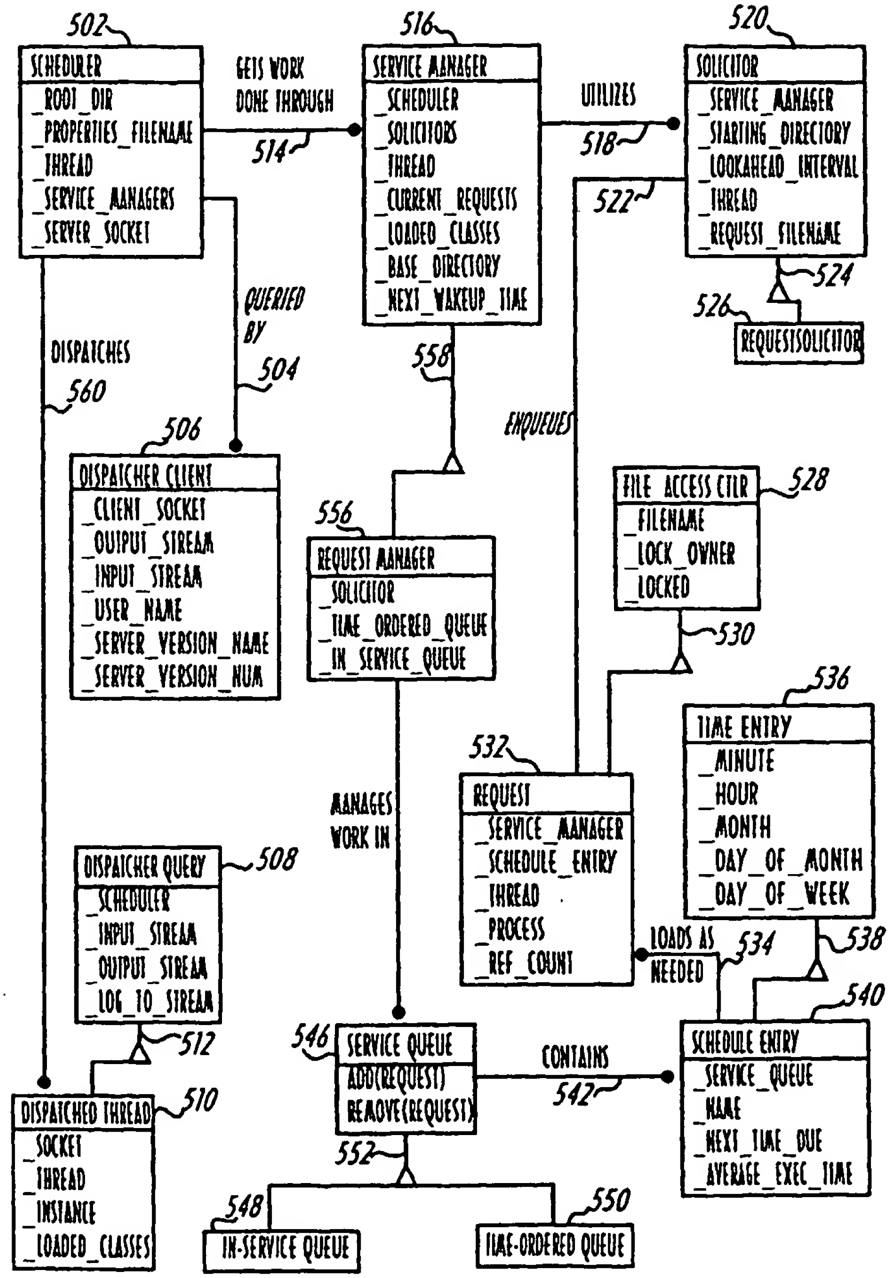
502
SCHEDULER
_ROOT_DIR
_PROPERTIES_FILENAME
_THREAD
_SERVICE_MANAGERS
_SERVER_SOCKET
GETS WORK DONE THROUGH
514
516
SERVICE MANAGER
_SCHEDULER
_SOLICITORS
_THREAD
_CURRENT_REQUESTS
_LOADED_CLASSES
_BASE_DIRECTORY
_NEXT_WAKEUP_TIME
UTILIZES
518
520
SOLICITOR
_SERVICE_MANAGER
_STARTING_DIRECTORY
_LOOKAHEAD_INTERVAL
_THREAD
_REQUEST_FILENAME
522
524
526
REQUESTSOLICITOR
QUERIED BY
504
DISPATCHES
560
506
DISPATCHER CLIENT
_CLIENT_SOCKET
_OUTPUT_STREAM
_INPUT_STREAM
_USER_NAME
_SERVER_VERSION_NAME
_SERVER_VERSION_NUM
558
ENQUEUES
556
REQUEST MANAGER
_SOLICITOR
_TIME_ORDERED_QUEUE
_IN_SERVICE_QUEUE
FILE_ACCESS CTLR
_FILENAME
_LOCK_OWNER
_LOCKED
528
530
536
TIME ENTRY
_MINUTE
_HOUR
_MONTH
_DAY_OF_MONTH
_DAY_OF_WEEK
532
REQUEST
_SERVICE_MANAGER
_SCHEDULE_ENTRY
_THREAD
_PROCESS
_REF_COUNT
MANAGES WORK IN
DISPATCHER QUERY
_SCHEDULER
_INPUT_STREAM
_OUTPUT_STREAM
_LOG_TO_STREAM
508
LOADS AS NEEDED
534
538
540
512
DISPATCHED THREAD
_SOCKET
_THREAD
_INSTANCE
_LOADED_CLASSES
510
546
SERVICE QUEUE
ADD(REQUEST)
REMOVE(REQUEST)
CONTAINS
542
SCHEDULE ENTRY
_SERVICE_QUEUE
_NAME
_NEXT_TIME_DUE
_AVERAGE_EXEC_TIME
552
548
IN-SERVICE QUEUE
550
TIME-ORDERED QUEUE

FIG. 5

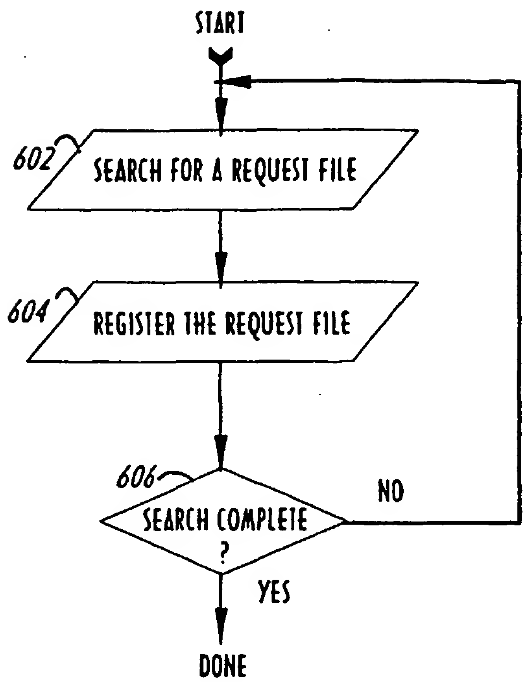
START
602
SEARCH FOR A REQUEST FILE
604
REGISTER THE REQUEST FILE
606
SEARCH COMPLETE
?
NO
YES
DONE

FIG.6

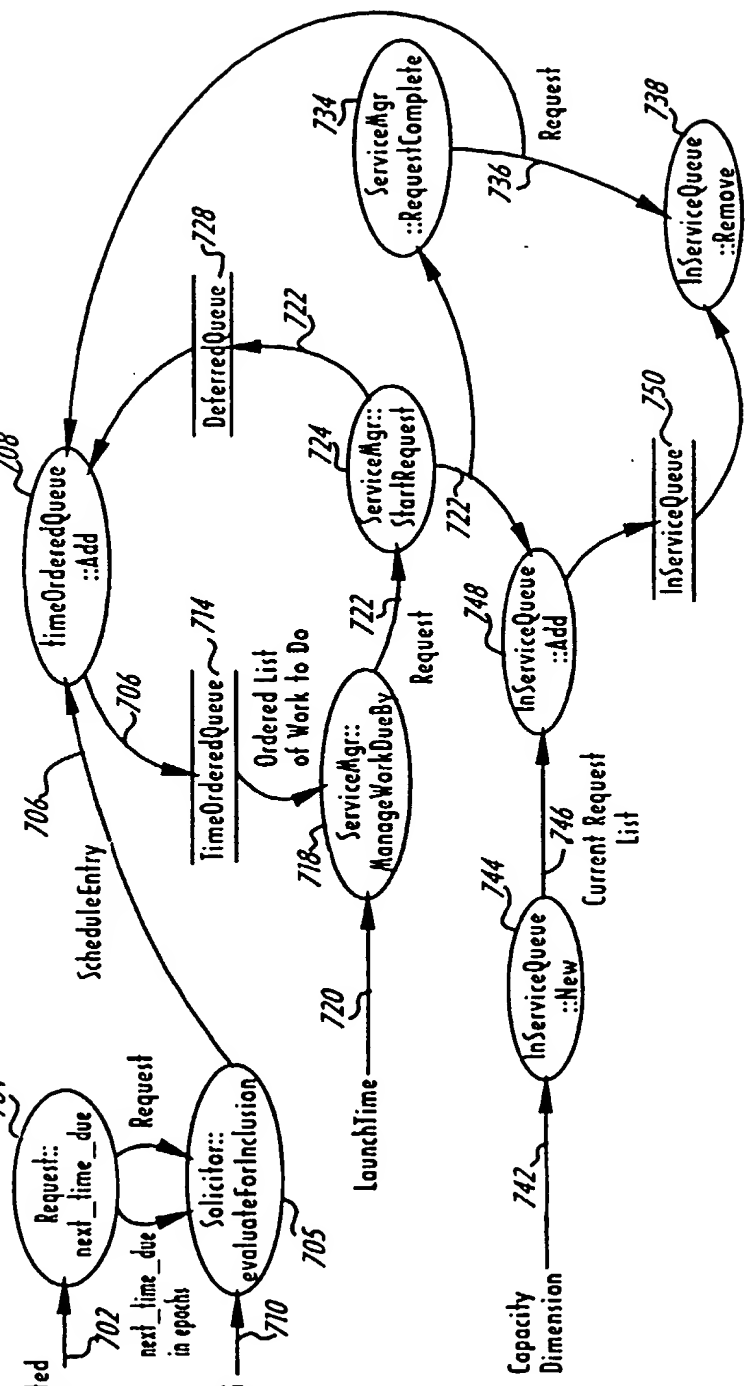
Requested File
702
next_time_due in epochs
QUEUEING INTERVAL
710
Request::next_time_due
704
Request
Solicitor::evaluateForInclusion
705
ScheduleEntry
706
TimeOrderedQueue::Add
708
706
TimeOrderedQueue
714
Ordered List of Work to Do
ServiceMgr::ManageWorkDueBy
718
LaunchTime
720
722
Request
ServiceMgr::StartRequest
724
722
DeferredQueue
728
722
ServiceMgr::RequestComplete
734
736
Request
InServiceQueue::Remove
738
InServiceQueue
750
InServiceQueue::Add
748
Capacity Dimension
742
InServiceQueue::New
744
746
Current Request List

FIG. 7